Expertise
and insight
for the future

Aleksi Alatalo

Serverless IoT Platform

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Electronics

Bachelor's Thesis

30 May 2020

Author Title Number of Pages Date	Aleksi Alatalo Serverless IoT Platform 22 pages 30 May 2020
Degree	Bachelor of Engineering
Degree Programme	Degree Programme in Electronics
Professional Major	Electronics
Instructors	Kalle Ahola, Project Manager Anssi Ikonen, Senior Lecturer

The purpose of this project was to design software architecture for an IoT platform for data logging. The platform was required to be cloud based, secure, easily configured and support dynamic data.

The platform is made of three parts, sending device, server and user interface. For the device part, a specification for the message format and protocols to be used was decided. For the server part, suitable components and services was chosen from the selected cloud service provider Microsoft Azure. Azure Functions was used as the compute resource and as way to logically connect the different server components together. MongoDB was chosen as the database; this supports dynamic data easily compared to traditional SQL databases. For the user interface a SPA, the single-page application, web app was chosen, as SPA architecture meshes well with the platform server architecture.

The resulting architecture meets all of the requirements. Security requirements are met with use of secure Azure components. Requirements for ease of configure and dynamic data is achieved by leveraging the abilities of MongoDB to store dynamic data and automatically configure by the inputted data.

The designed platform shows one way to collect IoT data and there are many existing systems for this. The platform is designed to be simple to use with minimal configure.

Keywords	IoT, Azure, Azure Functions, MongoDB

Tekijä Otsikko Sivumäärä Aika	Aleksi Alatalo Serverless Iot Alusta 22 sivua 30.5.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Sähkö- ja automaatiotekniikan tutkinto-ohjelma
Ammatillinen pääaine	Elektroniikka
Ohjaajat	Projekti Päällikkö Kalle Ahola Lehtori Anssi Ikonen

Tämän projektin tavoitteena oli suunnitella ohjelmisto arkkitehtuuri IoT datan keräys järjestelmälle. Järjestelmän vaatimuksina oli olla pilvi pohjainen, tietoturvallinen, helposti määritettävä ja tukea dynaamista dataa.

Järjestelmä koostuu kolmesta osasta, lähettävä laite, palvelin ja käyttöliittymä. Laite osalle määritettiin viestin formaatti ja käytettävät protokollat. Palvelimelle valittiin sopivat komponentit valitun pilvipalvelu tarjoajan Microsoft Azuren valikoimasta. Azure Functions valittiin käytettäväksi laskenta palveluksi ja yhdistämään kaikki palvelimen osat toisiinsa. MongoDB valittiin tietokannaksi. Se tukee helpommin dynaamista dataa verrattuna perinteisiin SQL tietokantoihin. Käyttöliittymäksi valittiin yhden sivun SPA web applikaatio, SPA arkkitehtuuri sopii hyvin yhteen palvelin arkkitehtuurin kanssa.

Lopullinen arkkitehtuuri tavoittaa kaikki asetetut vaatimukset. Tietoturvallisuus käyttämällä turvallisia Azure komponentteja. Helposti määritettävä ja tuki dynaamiselle datalle saavutettiin hyödyntämällä MongoDB:n ominaisuuksia ja automaattisesti määrittämällä järjestelmä syötetyn datan mukaan.

Tämä on yksi tapa kerätä IoT dataa, tähän on useita valmiita ratkaisuja. Kehitetty järjestelmä on suunniteltu olemaan yksinkertainen käyttää minimaalisilla määrityksillä.

Avainsanat	IoT, Azure, Azure Functions, MongoDB

Contents

List of Abbreviations

API	Application Programming Interface, a defined interface used to interact with other programs.
DB	Database
REST	REST.
IoT	Internet of Things.
JSON	JavaScript Object Notation

1 Introduction

This thesis is about designing an architecture for a serverless IoT platform hosted in the cloud. Work was done as an internal development project in the company I work. The team I am part of is focussed on automation of testing and quality assurance. In many of the team's projects there is a need to store data in a database and preferably in the cloud.

The purpose of this project was to design and develop a system for data logging, as most of the cloud storage needs for the projects are for data ingress from different systems. The system should be able to be used in many kinds of projects with minimal, preferably none, modifications and configuration to the system itself. In addition, the data logging to the system should be simple to add to those projects.

The scope of this thesis is limited on the design and architecture of the platform, with focus on the parts hosted in the cloud.

2 Requirements

In this chapter the requirements for the platform are introduced and explained.

The goal of this project was to make a platform for storing measurement data in a database for use in company. Typically, each project has its own custom solution for this problem and barrier for doing quick measurement setups is the database aspect of it. Plan was to make a platform that could be used on multiple projects and ad-hoc measurement setups.

Visualisation and analysis of the measurements with project specific solutions is another common point between different projects. The platform must have also capability for basic visualisation and analysis, also data must be able to be exported to more robust analysis tools.

The main requirements for the platform are the following:

- Cloud based. The platform should be hosted on a cloud service provider, Microsoft Azure or Amazon AWS for an example. This removes the need to manage servers. Cloud service providers have ready-made solutions for many problems. Scaling of the platform is also simplified, as extra server capacity is easily and readily available on the providers.
- Time-series like data. The data the platform is required to handle is time-series like, as in all data rows correspond to a point in time, for example temperature at different times of a day.
- Data is stored and is query able. The platform is required to store all inputted data for a specified time period. The data is also needed to be accessible and able to be exported in .csv format for example.
- Easily configurable for different applications and measurement setups. The platform should be able to handle different kinds of measurement devices and environments at the same time, for example an embedded device measuring temperature of a building and a PLC monitoring a factory production line.
- Support for plugins. For future expandability and per project customizability, a plugin architecture from the start will make implementation of these easier.
- Dashboard style interface. The platform should have a dashboard, where current values and trend lines of measured values are shown. In addition, a basic analysis tool.
- Multi-tenant support. The platform should be able to be used by multiple different clients at the same time.
- Secure. All connections must be secure and in case of multi-tenant, no crosstalk between the clients, each client should be isolated.

3 Technologies Used

In this chapter the most relevant tools and technologies used in the platform are introduced and briefly explained. In chapter 3.1 MongoDB is introduced and how it differs from a traditional SQL type database. In chapter 3.2 Azure, the chosen cloud service provider, is introduced and from the used services the most important to the platform are also introduced. In chapter 3.3 is the chosen frontend framework Angular and in chapter 3.4 is the development environment briefly.

3.1 MongoDB

MongoDB is a NoSQL document database program developed by MongoDB Inc. MongoDB can be run on the most common operating systems, Mac, Windows and Linux.

Versions after October 16, 2018 are published under Server Side Public License SSPL and its free community version is source available. [1.]

3.1.1 NoSQL Versus SQL

NoSQL is used as a term to indicate non-relational databases, which means databases that store data in other way than relational tables. There are several types of NoSQL databases, four most popular are: document, key-value, wide-column and graph. [2.]

Relational tables have a fixed schema, meaning that table columns are fixed, and each row must have all these rows. In relational tables, a table row may be linked to other rows in other tables and this is in the database level. For example: in an orders table an orders customer id column is a foreign key linked to customers table where customer id is a primary key. Customer's information, delivery address etc., are only stored in the customer table and when an orders delivery address is needed, it must be joined with the customer. [2.]

Document databases store the data in form of documents, each row is a document and the structure of the document can wary between the documents in the same table. Documents are often directly mapped to objects used in programming. The previous orders example could be simplified with document database by storing all the information an order needs to a single order document. MongoDB uses BSON, binary version of JSON. [1;2.]

3.2 Azure

Azure is a cloud computing service by Microsoft. The platform uses Azure as its cloud service provider. Multiple different services are in use and the most important ones for the platform are introduced in this chapter.

3.2.1 Azure IoT Hub

Azure IoT Hub is a heavy duty IoT message hub. It is a managed service, that can handle millions of connected devices with millions of events per second. It is guaranteed to be 99.9% available. A device can connect to it using one of the following protocols:

- HTTPS
- AMQP
- AMQP over WebSockets
- MQTT
- MQTT over WebSockets

Microsoft also provides client libraries to ease the development of connecting to the IoT hub for most common platforms and programming languages. [3.]

IoT Hub is priced in units, that have a fixed count of messages per day that can be sent or received. Each IoT Hub subscription can comprise of multiple units, increasing the count of allowed messages. There are also other quotas that limit the use, mainly the messages per second, these can also be increased by adding more units to the subscription. [4.]

3.2.2 Cosmos Db

Azure Cosmos DB is a managed multi-model database service. Multi-model meaning that the database is not fixed to a single model, like table, it can be configured to be a document, a key-value or one of the other supported models. [5.]

Cosmos DB is globally distributed, a user can choose multiple regions for their subscription to further secure their data and lower latencies. [5.]

Cosmos DB is priced by Request Units per second and storage space used hourly. Request Unit is abstraction of database operations: insert, read, update etc. Request Units are billed by the hourly average, so a subscription for fixed amount of Request Units per second per hour can still scale for instant peak demands multiple times of the subscribed amount. [6.]

3.2.3 Azure Functions

Azure Functions is an open-source service for serverless computing, it provides a platform for users to run their code on with ready-made application infrastructure. Functions are event based, users code is run when the configured trigger, which can be a HTTP request or any of the other supported trigger types, is triggered. Users can program their functions on C#, Java, JavaScript, Python or PowerShell. [7.]

Functions are deeply integrated to other Azure services with the triggers and bindings, bindings are way to connect to another resource as inputs and outputs. [8.]

Functions are priced by number of total executions and execution time. Execution time is measured by the memory used times the execution time in milliseconds. [9.]

3.2.4 Durable Functions and Entities

Functions are stateless by design, as different Function instances can be run on different virtual machines at the same time and they do not share resources. A single function execution runtime is also limited. For some task's persistence of state or longer runtime is needed and for that is the Durable Functions framework built on top of Functions. [10.]

Durable Functions allows the development of stateful workflows on the Functions platform. This is achieved with orchestrator functions, a function with a special kind of trigger and bindings. Other function types are activity and client functions. Activity functions are like the normal functions and all actual work should be made in them. Client functions are the only thing that can trigger an orchestrator function. [11.]

Orchestrators work by calling activity functions, during runtime of activity functions the orchestrator is stopped, and its state is stored. After activity finishes its execution, its return value is stored, and the orchestrator is restarted. When the restarted orchestrator reaches a call to activity function, instead of calling it, it uses the stored return value and continues. This is repeated until the orchestrator finishes. [10.]

Durable entities are fourth type of Durable Function, they are like objects, as in they have methods that can be called and there can be multiple instances of an entity class.

Durable entities can be signaled one directionally from a normal function and called with a return value from a durable function, when called they act like an activity function. A durable entity can also signal other entities. [12.]

3.2.5 SignalR

SignalR is an opensource service for real-time communication, mainly used in web development for server to client messaging. It has easy to use libraries for both client and server side, making integrating of real-time functionality for a web app simple. [13.]

SignalR supports several different methods for real-time communication in order of preference: WebSockets, Server-Sent events and Long Polling. It chooses the best method both server and client support during handshake. [13.]

Azure SignalR service is a managed standalone SignalR server. As real-time methods rely on connections being open for long time, an Azure Function app on its own can not support them, as a Functions runtime is limited. The Azure SignalR service is a good way to bring real-time capabilities to a Function app, it being a separate service and not bound to the limits of Functions. [14.]

Azure SignalR service is priced with a flat per day price for a unit, each unit has a limit of maximum concurrent connections and messages per day. Several units can be added together summing their limits. [15.]

3.3 Angular Framework

Angular is an open source SPA web framework. SPA, or single-page application, is a type of web application, where the rendering, the generating of the HTML code, is done at the client using JavaScript. When a SPA site is accessed, single HTML file is loaded and that links to the JavaScript modules housing the actual site. When the modules are loaded and executed, the site is rendered, making it a single-page as only one page is actually accessed at the server. [16.]

Other ways to render webpages is static files or server-side rendering. At the server-side SPA is the same as static files.

3.4 Development environment

3.4.1 Visual Studio Code

Visual Studio Code is an open source text editor by Microsoft. It has many features for programming, debuggers, syntax highlighting etc. It is extendable with support for different programming languages and extra features. [17.]

All the programming of the platform is made on Visual Studio Code. Azure extensions are installed for support to used Azure services.

3.4.2 Git and GitHub

Git is a version control system for source code. It is intended for use with text files, but all file types are supported. GitHub is a hosted service for git.

The platform's code repository is hosted in GitHub.

3.4.3 C# and .NET CORE

C# is type-safe object-oriented programming language. Its syntax is similar to C, C++ or Java. It is of higher level than C++ and is compiled language. [18.]

.NET CORE is an open source framework for general purpose software development, C# is used to write applications to it in addition of Visual Basic and F#. [19.]

Azure functions use .NET CORE framework and can be programmed with C# [7].

3.4.4 TypeScript

TypeScript is programming language based on JavaScript, it adds support for types to JavaScript, making type safe programming possible, as JavaScript is very loose regarding types. TypeScript code is compiled to standard JavaScript for execution, making it compatible with all browsers supporting JavaScript. Angular applications are programmed with TypeScript [16]. [20.]

4 Software Architecture

In this chapter the software architecture of the platform is explained: how it meets the requirements introduced in chapter 3 and how tools and technologies introduced in chapter 2 are used to solve architecture problems. Chapter 4.1 is introduction to the concept of serverless. In chapter 4.2 the high-level overview of the entire platform is explained. In chapter 4.3 we focus on the server side of the platform and in chapter 4.4 on the user interface side.

4.1 The Serverless Concept

Serverless as a term is bit misleading, there are still servers, but they are hidden behind an abstraction layer. The serverless service provider takes care of the server hardware and the VM: s that run on them, the user only needs to worry about the software that is run on the service.

Serverless services are usually billed by the actual use instead of fixed cost per time period, that is common for normal cloud-servers. Serverless services can also scale automatically depending on the demand. For example: A software that is needed once a week with a very high demand. With traditional cloud-servers, a powerful and expensive server would be needed to meet the high demand, but it would be idle for most of the time. The serverless version would simply lay dormant when not in use and then scale up to the demand and would be billed only for the used capacity.

4.2 High Level Overview

The platform consists of three parts: device, server and the user interface web-app. All of these are also run on different environments. Server has API: s for device and web-app connectivity. Below in the figure 1 is the figure of the platform's architecture, all of the elements inside the "Azure" box make the server part.

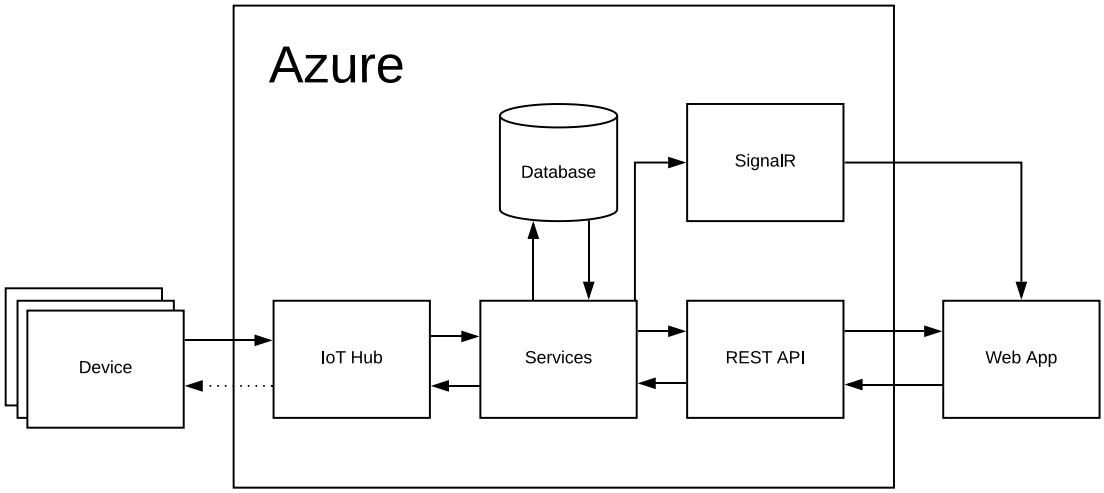

Figure 1. Architecture of the platform

The platform is separated into virtual containers, known as accounts. User is a registered user-account that has access to the platform. Each account has its own devices and users. In the database data from different accounts is stored mixed, but user from account A cannot see anything from account B. Admin-level users have manage and have access to all accounts.

4.3 Server

The server part, the backend, of the platform is the most complex part of the platform. It processes data from the devices, manages the platform as whole and handles user interface requests. The backend comprises of four parts, IoT Hub, database, REST-API and the services that connect these parts together and houses the necessary logic. Architecture of the server part and how its parts are connected together is pictured in the figure 1.

The backends architecture is of serverless design, as all of the used Azure services can be categorized to be ‘serverless’. Reason for choosing a serverless design is to be able to fully leverage the potential of cloud-computing services, with serverless the platform can scale up easily and without needing to procure and set up new servers, the only thing needed to do is allow the platform to use more money.

4.3.1 IoT Hub

Azure IoT Hub was chosen as the device facing API endpoint service as it encapsulates all the required functionality in one package. The IoT Hub also features several inbuilt ways to connect it to the rest of the Azure ecosystem, specifically the Event Hub endpoint, Event Hub is an event queue service, which is used in this software as the preprocessing message queue. [3.]

Devices must be registered to IoT Hub to allow communication with it, for this IoT Hub has a REST API. Device management part of the platform uses this interface to automatically register a device when adding a new device to the platform. [3.]

When a device wants to send a message to the server, it first connects to the IoT Hub using one of its supported protocols. In the case of HTTP, there is no need for a separate connections step, each message has the necessary authentication information embedded in it. Then the device sends the message(s) to the IoT Hub. Each message has in addition of the payload itself, the id of sender and metadata about the content of the message, for example what type of message it is. Message is then added to the Event Hub queue for processing.

IoT Hubs Event Hub queue stores the messages for up to seven days and as the IoT Hub is a separate component from the rest of the server, new messages can still be received normally even if the rest is on maintenance.

4.3.2 Database

Due to the dynamic nature of the platform, particularly the telemetry data from the devices, which can have varying number of fields with different datatypes, traditional SQL relational databases are not optimal choice. The telemetry can be stored in its dynamic

form in a relational database, but the resulting query and insert operations would be very complex, needing multiple joins to produce a single telemetry time point. In contrast with a NoSQL document database, like the Mongo Db used in this project, a single telemetry time point is a single document with all the fields in it.

Azure Cosmos Db is used as the database server, it is used with the Mongo DB API. By using Mongo Db API instead of the native Cosmos Db API, the used database server can be easily changed to a different Mongo DB compatible server without changes to the code.

By default, Mongo DB collections, collection in SQL terms would be analogue to a table, are indexed by the document id field. Document id is the only required field in a document, which uniquely identifies it inside the database. Additional indexes can be configured to optimize a collection. For example, in the platform's telemetry collection a composite index, index made of several fields, made of the device id and time stamp fields in that order. In this configuration the documents are sorted first by the device id and then by the time stamp, this ensures most efficient queries of telemetry data

4.3.3 Services

The platform requires server-side compute capabilities to handle the device messages, interface with the database, react to requests from the REST API and so on. Azure Functions provide the serverless compute the platform needs. Functions are also well connected to other Azure services and with their HTTP-trigger they can also act as the REST-API end points.

Device-group is a way for user to sort devices to different groups as they wish, this is purely for user interface use. Device-groups contain links to devices and sub-device-groups. All of the single account's device-groups are sub-groups of single a group and as all of the account's devices are linked to one of the groups, this forms a device-tree. Dashboards in the user interface can be made for a group, this can show data from any of the devices linked to the group or its sub-groups.

Durable entities from the Durable Functions framework are used to manage writes to the database. Many of the used write operations are updates, modifying of existing entries.

A call for example to move a device from one group to another, would first need to fetch the corresponding group documents from the database, check that the groups exist and the device is even linked to the from-group, modify them and then update the database with the modified documents. This is possible in a normal function, but if two calls to move the same device is made at the same time, those function executions could be executed at the time and could result in unwanted results. Instead, all of the documents, that can be updated in this way, have durable entity copies of them, devices have device-entities, device-groups device-group-entities and so on. All of the update logic is inside these entities and all of the writing to these documents is made through the entities. As all entity method calls and signallings are processed through a queue, this ensures that all update operations are made one at time and in order. The same device-group example with entities: first check that the groups and device exist from a local cache of valid ids, signal the from-group-entity with the device and target group ids, the entity checks that the device is linked to it, removes the link and updates the database, signals the to-group-entity with the devices id, to-group-entity adds the link and updates the database.

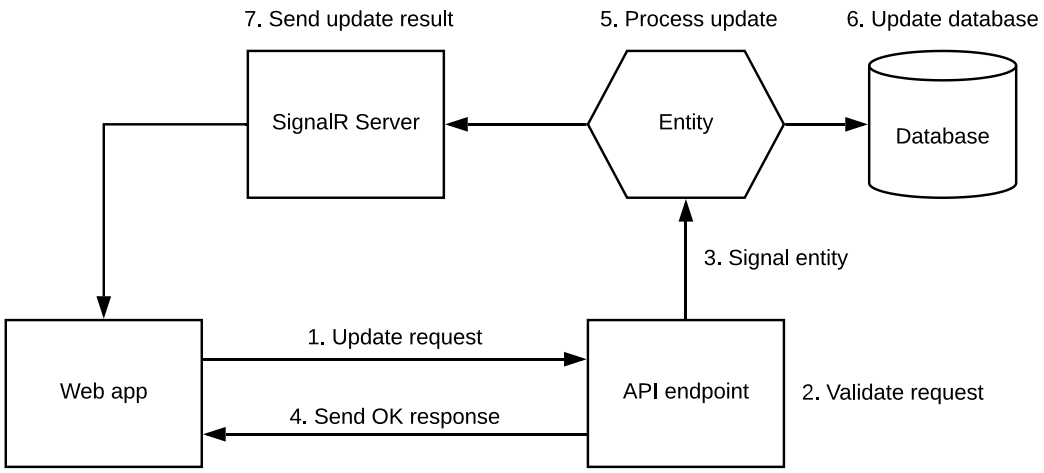

Figure 2. Step-by-step of update via an entity.

Calls to functions, that start an update operation via an entity, return before the update operation has started, as the update is processed in the entity. If the update modifies something that should be indicated on the user interface, for example moving a device from one device-group to another, the user interface should somehow be notified of the result of the update. SignalR is used for this and the process is pictured in the above figure 2 step-by-step. When an entity is updated, it sends the update result to the SignalR

server, which then forwards it to all subscribed listeners. The user interface listens on update messages from all entity-based operations, but it can only receive messages that correspond to the current users account. This also means that if user A modifies something, then user B also receives the update at the same time. Real-time telemetry is also sent via SignalR.

The message routing and pre-processing, platform and device management are basic functionalities of the platform, rest of the program is capsulated in plugins. Plugins can be managed on per device basis, each plugin can have device specific options and can be enabled or disabled as well. Plugins can be categorized to three different types, internal, external and frontend only. Internal plugins are baked in the code of the main program and external plugins are outside of the program, messages are routed to them via HTTP. Frontend only plugins are implemented only on the frontend web-app, they use the basic capabilities of the platform. Both internal and external may also have their own frontend parts.

4.3.4 REST API

The server has a REST API intended for use by the frontend. API has endpoints for account, user, plugin and device management, telemetry query and also plugin specific endpoints.

The API is secured with JWT authentication. Requests to an endpoint requiring authorization must have a JWT security token in their headers. Tokens have users claims encoded in them, in the case of the platform: user level, user id and account id. Tokens are cryptographically signed by their issuer. Modern SSO used by many sites, for example signing with Google account to another site, uses JWT as their authorization method.

Each endpoint has varying level of security. *Anonymous* for login related endpoints, as the user is not yet authenticated. *Standard* for authenticated users, this is for resources common for all accounts. *Account level* for account specific resources, users account is compared to the account of the resource requested. *Account level* with elevated access, this is for update operations, standard user has no authorization to update. *Admin level*, this is the highest level, all operations are allowed.

4.4 Device

The device part of the platform is not a specific device or software stack, it is specification of how to connect to the IoT Hub and message formats. The device can be a PLC, embedded device, a pc or something else, the software can be written in C, Python, LabView etc., if it can send messages to the hub in the correct format, anything goes.

For message format specification there are two layers: the IoT Hub message format and inside it is a payload, which is in format specified by the platform. Inside IoT Hub message, besides the payload, are also metadata flags. these are also specified by the platform.

Microsoft provides IoT Hub programming libraries for several programming languages to make integrating connection IoT Hub easier to programmers. If there is no ready-made library for a programming language, the connection can still be achieved by following the specifications.

During development of the platform, a simulated device was used.

4.5 User Interface

The user interface, or frontend as it is commonly called in web software development, is not a focus of this thesis and most of the development and design were made after the thesis work. In this chapter a general overview of the frontend is given.

For frontend a web app was a logical choice and a SPA type especially, as it moves the web page rendering from a server to the client web browser, this fits the serverless architecture of the platform. Angular was chosen as the SPA framework, as I have previous experience with it.

The frontend functions as a user-friendly wrapper on the server REST-API.

Many of the data consumption plugins are frontend only, this means that their state doesn't influence the server-side functionality at all. They rely on the basic data-access methods of the server.

5 Design Process Example

In this chapter we take a deeper look on the core of the platform, the message pipeline, and on its iterative design process from the simple proof of concept version to its current efficient version.

5.1 Message pipeline

Task of the message pipeline is to parse and handle all the server bound messages from the devices. The pipeline is required to be:

- Be secure. As one end of the pipeline is open to the internet, it must be secure. IoT Hub was mainly chosen for its security, as security can be hard to implement and it's easier to use existing solutions. All of the security aspects of the pipeline are handled by the IoT Hub.
- Able to handle all the different kinds of messages. The platform is planned to be support multiple kinds of messages, in this stage of the project, only the basic telemetry messages are supported. But the pipeline should be easily modified to support other types of messages also.
- Robust. Storing of the data, which is one of the tasks of the pipeline, is the most important part of the platform to work correctly and reliably. Data sent to the platform is expected to be stored at least in some form. This is partly achieved by storing of all the messages received by the IoT Hub in their raw form in a blob-storage.
- Keep track of different data fields found on telemetry per device. As the telemetry can contain any number of fields with different datatypes and a single telemetry message from a device might not contain all the different fields that device sends. There is a need to store the sent field names per device to the database.
- Plugin architecture. The pipeline should support the plugin architecture, as there might be need in future for example to route the messages to an external service.

- As efficient as possible. The message pipeline must be as efficient as is reasonably possible, as it is the most compute heavy part of the platform and has highest traffic. As compute is pay per use, cost of compute is directly related to efficiency here.

5.2 Version 1: Simple

Simple, or the proof of concept, version was used to test the pipeline from a device to the database.

The pipeline starts from the IoT Hub, in this stage is the actual endpoint to which the devices send the messages. IoT Hub first authenticates the sending device, only devices registered to the IoT Hub can send messages, then the messages are inserted to the Event Hub queue. A function with Event Hub trigger reads the queue.

Event Hub has multiple queues, that are used to balance load on their consumers. Queue is used to buffer the messages in FIFO, first in fist out, manner. An Event Hub trigger on a Function is triggered if any of the queues has messages in it, even one message is enough to trigger. The triggered function then takes batch of messages, up to configured maximum limit. In practice this means that with high loads, the triggered function is more efficient in the meter of function calls per message. And in lower loads messages are processed with low latency, but worse efficiency.

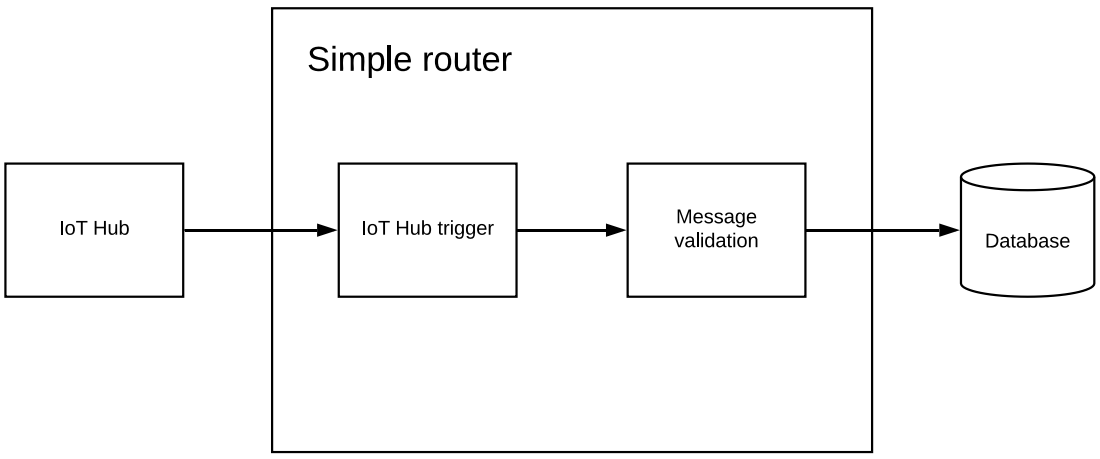

Figure 3. Architecture of message pipeline: the simple router

Rudimentary data validation is made for each message, making sure that the minimum required fields are met, device id and time stamp. And then the messages are inserted to the database. This simple architecture is pictured in the figure 3.

This is the most efficient and robust version of the pipeline, but it lacks the ability to handle different kinds of messages, track field names per device and no plugin support. These all could be made to the single function version, but as normal functions are stateless, each function run instance would have to make multiple database queries to handle the plugins and device operations. This would ruin the efficiency, as database compute time is more expensive and limited compared to functions.

5.3 Version 2: Message Router

This version was made to address the deficiencies of the simple version, namely support for different message types, plugins and tracking of telemetry field names. This was achieved using Durable Functions. Ability to store state inside a function reduces the extra database queries dramatically, instead of needing to access the database each run instance, a single read and writes only when needed is possible.

Durable entity classes were made for device and plugin. These map one on one to their respective database entries. They are initialized with a copy of the entry from the database and after that all write operations to the corresponding document are made only from the entity. This ensures that the entity and database version are always the same.

The same Event Hub triggered normal function has to be used as the starting point, as durable function orchestrators, orchestrator function is name for a function which keeps track of its state and can run for extended periods of time, cannot be started with the same triggers as normal function, they must be started from a normal function with a start orchestrator command and the initial data is passed with the start command.

After the message router orchestrator function is started, relevant device entities are called to update telemetry fields and get the active plugin ID’s from each device. After

that the plugins themselves are received from the plugin entities. With devices updated and plugins acquired, can the messages be routed according to message type and plugins in use.

Problem with this system is the number of function execution times needed for the whole process, a single message with a single plugin needs function executions, the initial normal function, first start of the orchestrator, call to device entity, first restart of the orchestrator, call to plugin entity, second restart. In addition to multiple function executions, between each the state of the orchestrator and entities is serialized to JSON and deserialized back, adding to required processing time. As the initial function triggers as fast as it can and starts a message router, this leads to each message being processed on its own message router instance and that is highly inefficient.

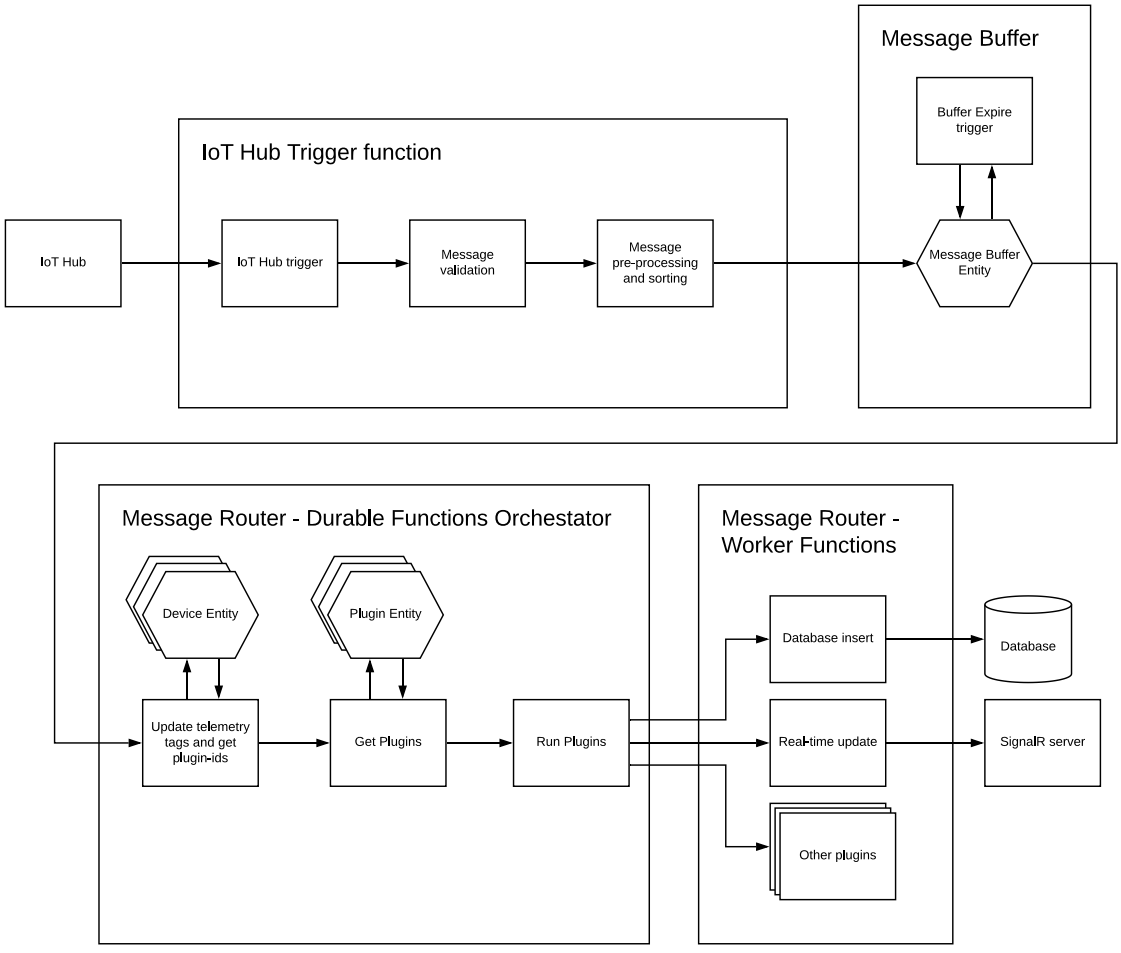

Figure 4. Architecture of message pipeline: the message router.

To remedy this a buffer entity was made to be between the trigger and message router. Trigger function sends the messages to the buffer entity and the buffer entity starts a message router and empties itself to it when it has reached maximum message count or oldest message has reached a time limit. This forces the message router to process messages in batches raising the efficiency. In the figure 4 is the final architecture of the message router version with the message buffer.

5.4 Version 3: Complex Device Entity

The previous version, the message router, works and meets all the requirements, efficiency being bit poor. But it is very complex and has too many parts, buffer being good example. This can be simplified greatly with the complex device entity version.

Idea of the complex device entity is to move the message processing to the device entity. All of the required information is stored there already, instead of updating telemetry fields, send all of the messages there. And with normal functions being able to signal entities, can the message router and buffer be cut away completely. With this in worst case two function executions are needed for each message and if message volume is high enough that trigger function receives multiple, can the device entities process the messages in batches also, raising the efficiency even more.

Figure 5 below shows the architecture of this version and flow of the data. The pipeline starts the same as the previous version, with the IoT Hub trigger -function. Difference being that after pre-processing and validation, instead of sending the messages to the buffer entity, they are instead sorted by the device the messages are from. After sorting, devices found in the messages corresponding device entities are then signaled with their new messages. The signaled entities then process the messages and run plugins inside a single function execution.

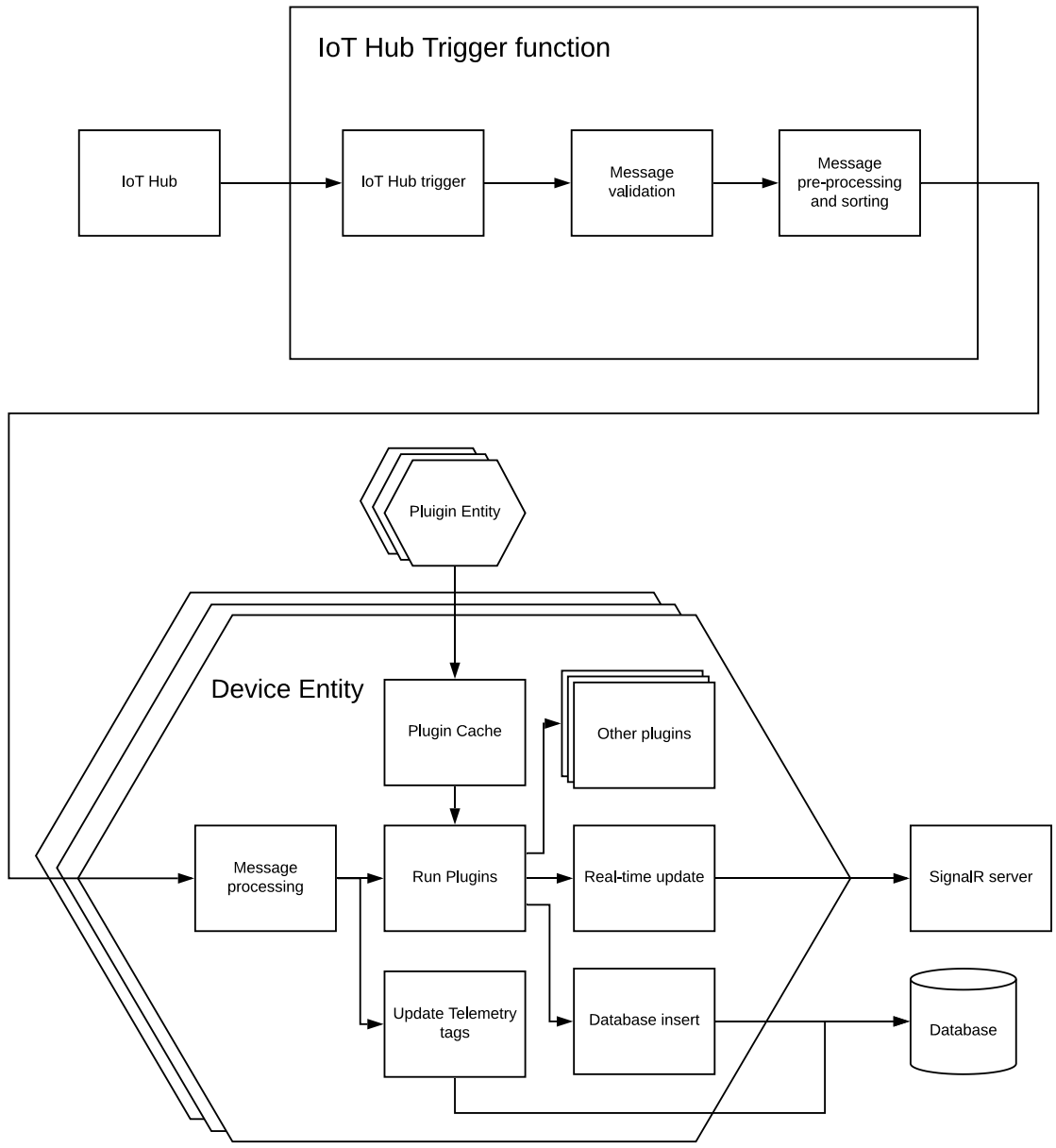

Figure 5. Architecture of message pipeline: the complex device entity.

Each device entity has a cache of relevant plugins, with settings specific for that device, stored inside. the entity has signals for plugin management, adding, removing and updating. Plugin entities are used to manage the plugin cache inside device entities.

This version is not without its potential problems. As the whole of the message processing in the device entity is computed during a single Function execution, it also limited by the maximum runtime limit of a single execution. The time required for IO operations with external services, such as database and HTTP-requests, is counted for the limit. If

the number of plugins and messages processed per function execution is too high, it may lead to exceeding of the runtime time limit. This stops the running execution and message processing is left to an unknown and possibly unfinished state. This can be alleviated with limiting the number of messages processed per function execution by batching the messages at the IoT Hub trigger -function to several entity signals. This is only a problem with very high message throughput limited to only few different devices, as messages are processed at each devices corresponding entity.

6 Conclusion

The goal of this project was to design an IoT data collection platform hosted on a cloud service provider. The platform is hosted on the Azure and uses several services from there for a serverless architecture. Serverless was chosen for its scalability potential. The platform was required to be easily configurable and able to ingest timeseries data with varying fields. By choosing a NoSQL database with no fixed schema, there is no need to configure the database to accept different kinds of rows and by keeping track of the different fields, the platform is automatically configuring itself in this regard.

There are multiple existing similar systems for IoT data logging, so the platform is not something revolutionary in this field. It is designed to be as simple to use as possible for basic data collection and visualization. It is not designed for heavy duty data logging, for example kilo Hertz range or above data logging frequencies.

The platform is in active development at the time of writing of this thesis. This thesis focusing more on the architecture and design aspect of the project. The message pipeline featured in chapter 5, is the only part of the platform that is “done” and is working as designed. Only after development reaches a point where most of the platform is working and it can be tested as a whole, can this architecture be truly evaluated.

References

1 MongoDB, What is MongoDB [online]
URL: https://www.mongodb.com/what-is-mongodb
Accessed 18.4.2020

2 Luke P. Issac, SQL vs NoSQL Database Differences Explained with few Example DB [online]. The Geek Stuff; 14.1.2014
URL: https://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/
Accessed 18.4.2020

3 Azure, IoT Hub documentation: What is Azure IoT Hub? [online] 8.8.2019
URL: https://docs.microsoft.com/en-us/azure/iot-hub/about-iot-hub
Accessed 19.4.2020

4 Azure, Azure IoT Hub pricing [online]
URL: https://azure.microsoft.com/en-us/pricing/details/iot-hub/
Accessed 19.4.2020

5 Azure, Cosmos DB documentation: Welcome to Azure Cosmos DB [online] 23.10.2019
URL: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction
Accessed 19.4.2020

6 Azure, Cosmos DB pricing [online]
URL: https://azure.microsoft.com/en-us/pricing/details/cosmos-db/
Accessed 19.4.2020

7 Azure, Functions documentation: Introduction to Azure Functions [online] 16.1.2020
URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview
Accessed 19.4.2020

8 Azure, Functions documentation: Azure Functions triggers and bindings concepts [online] 18.2.2019
URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings
Accessed 19.4.2020

9 Azure, Functions pricing [online]
URL: https://azure.microsoft.com/en-us/pricing/details/functions/
Accessed 19.4.2020

10 Azure, Functions documentation: What are Durable Functions [online] 7.8.2019
URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview
Accessed 19.4.2020

11 Azure, Functions documentation: Durable Functions types and features [online] 22.8.2019
URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview
Accessed 19.4.2020

12 Azure, Functions documentation: Entity functions [online] 17.12.2019
URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp
Accessed 19.4.2020

13 Microsoft, Introduction to ASP.NET Core SignalR [online] 27.11.2019
URL: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-3.1
Accessed 20.4.2020

14 Azure, What is Azure SignalR Service? [online] 13.11.2019
URL: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-overview
Accessed 20.4.2020

15 Azure, Azure SignalR Service pricing [online]
URL: https://azure.microsoft.com/en-us/pricing/details/signalr-service/
Accessed 20.4.2020

16 Angular, Introduction to Angular concepts [online]
URL: https://angular.io/guide/architecture
Accessed 29.4.2020

17 Microsoft, Visual Studio Code: Getting Started [online]
URL: https://code.visualstudio.com/docs
Accessed 29.4.2020

18 Microsoft, Introduction to the C# language and the .NET Framework [online] 20.7.2015
URL: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework
Accessed 29.4.2020

19 Microsoft, .NET Core overview [online] 26.3.2020
URL: https://docs.microsoft.com/en-us/dotnet/core/about
Accessed 29.4.2020

20 Microsoft, TypeScript [online]
URL: https://github.com/microsoft/TypeScript
Accessed: 29.4.2020